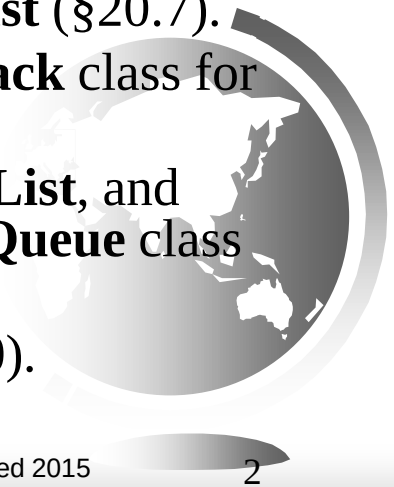
Chapter 20 Lists, Stacks, Queues, and Priority Queues

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy (§20.2).
- To use the common methods defined in the **Collection** interface for operating collections (§20.2).
- To use the **Iterator** interface to traverse the elements in a collection (§20.3).
- To use a for-each loop to traverse the elements in a collection (§20.3).
- To explore how and when to use **ArrayList** or **LinkedList** to store elements (§20.4).
- To compare elements using the **Comparable** interface and the **Comparator** interface (§20.5).
- To use the static utility methods in the **Collections** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§20.6).
- To develop a multiple bouncing balls application using **ArrayList** (§20.7).
- To distinguish between **Vector** and **ArrayList** and to use the **Stack** class for creating stacks (§20.8).
- To explore the relationships among **Collection**, **Queue**, **LinkedList**, and **PriorityQueue** and to create priority queues using the **PriorityQueue** class (§20.9).
- To use stacks to write a program to evaluate expressions (§20.10).

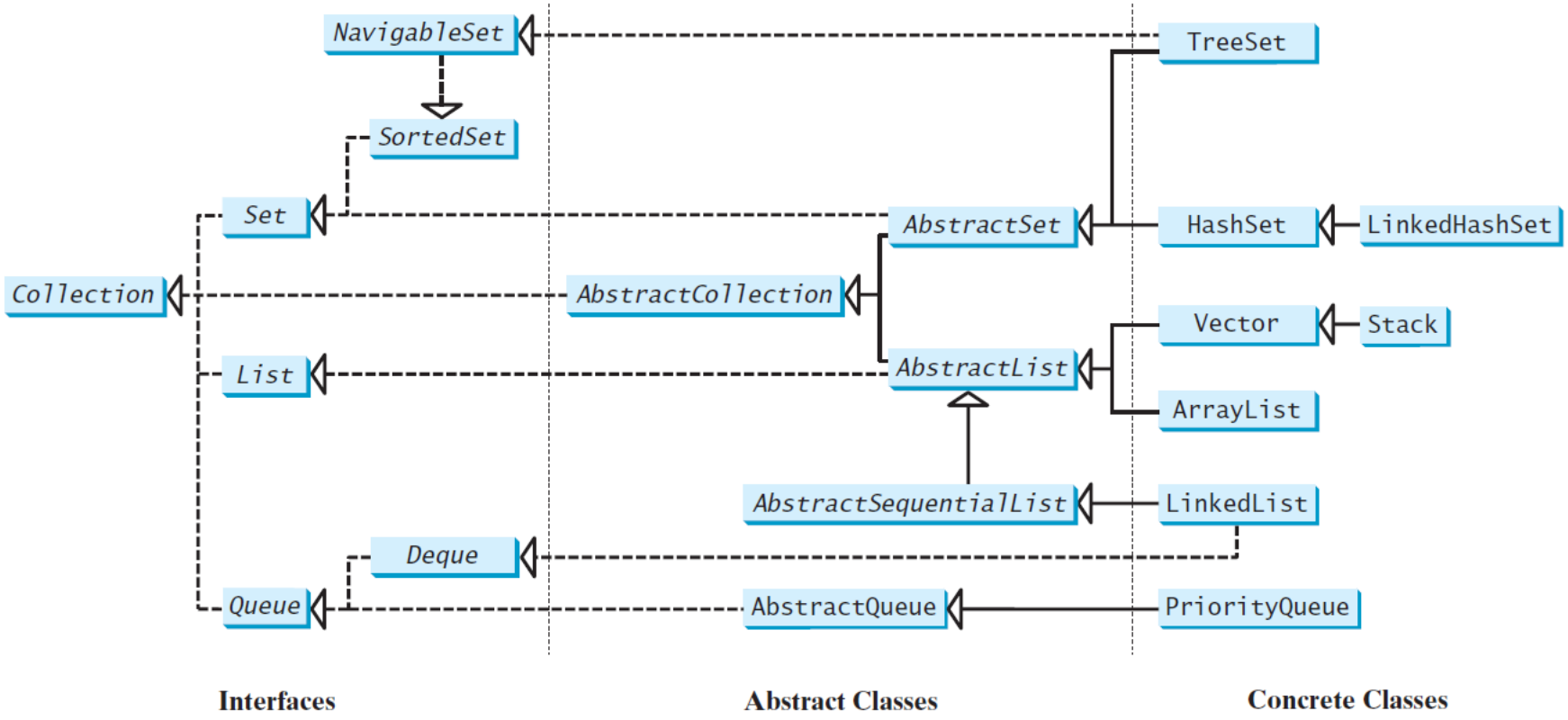


Java Collection Framework hierarchy

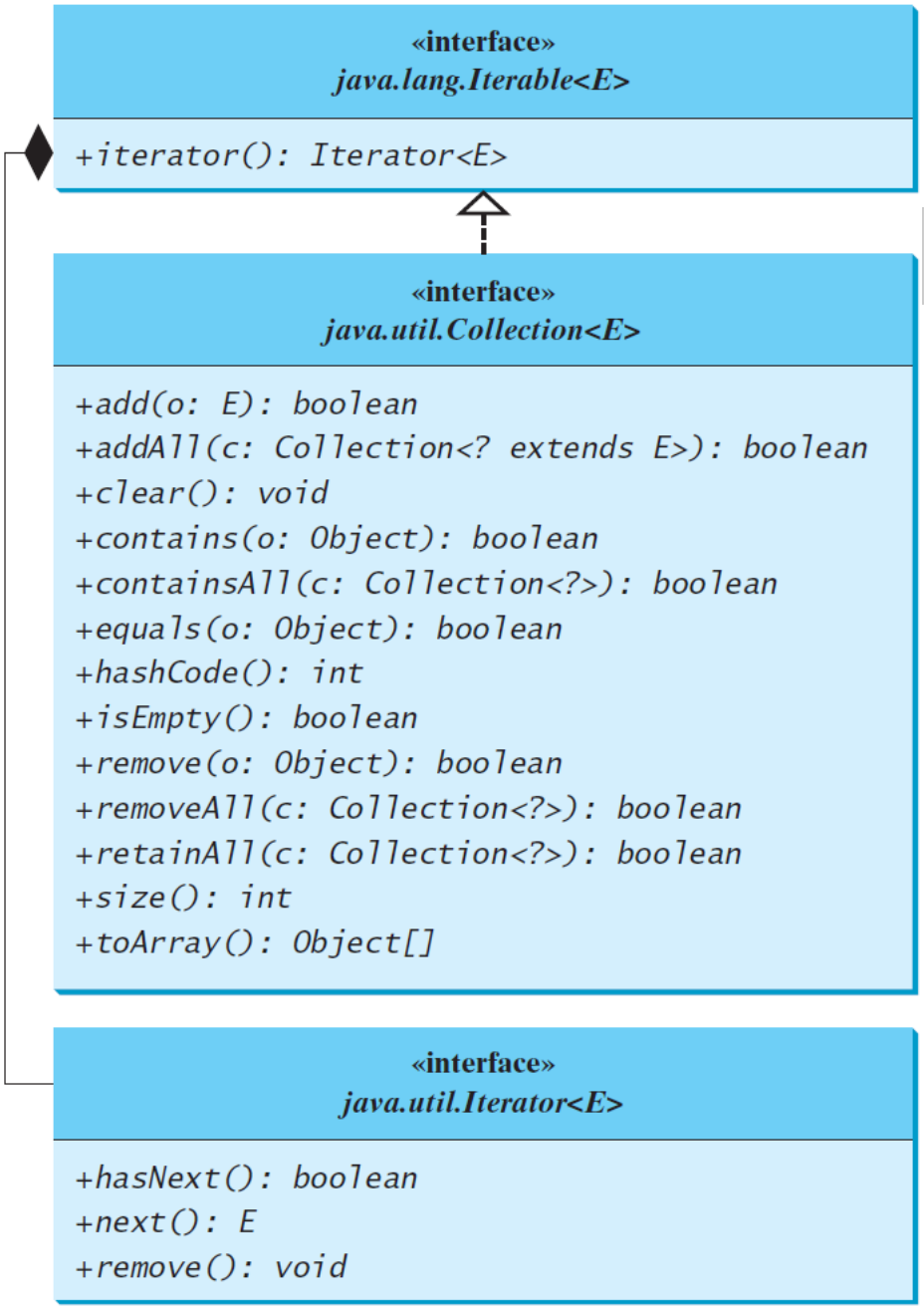
A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists*, *sets*, and *maps*.

Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



The Collection Interface



Returns an iterator for the elements in this collection.

The Collection interface is the root interface for manipulating a collection of objects.

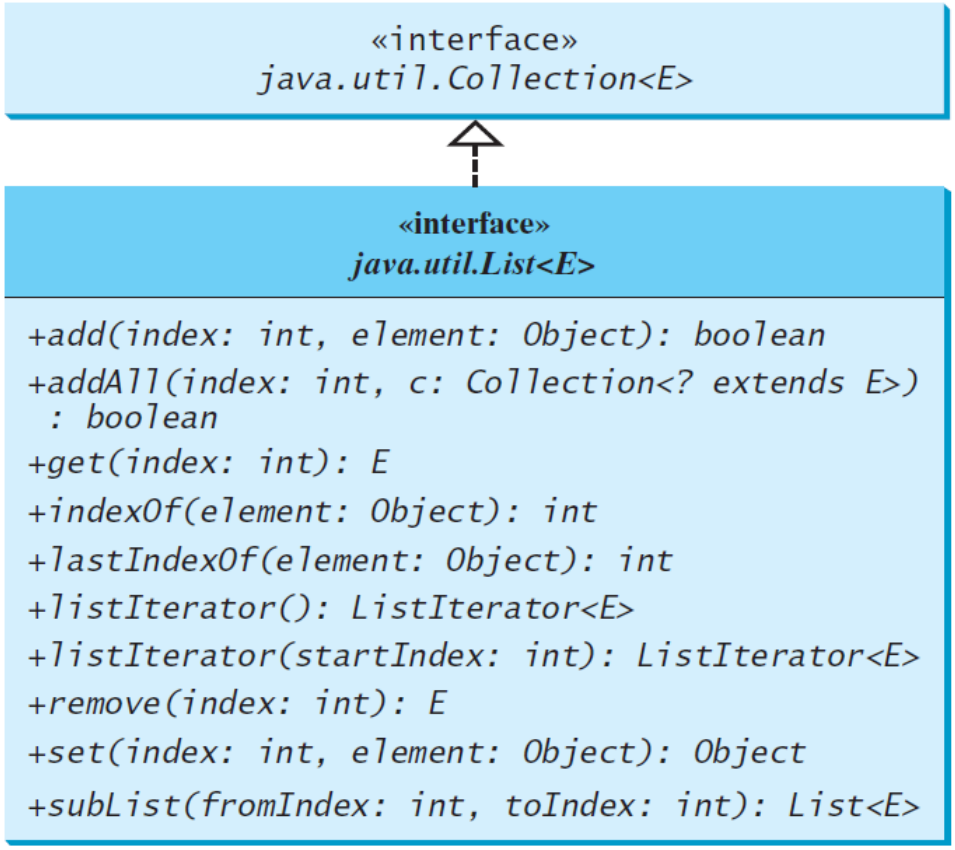
Adds a new element `o` to this collection.
Adds all the elements in the collection `c` to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element `o`.
Returns true if this collection contains all the elements in `c`.
Returns true if this collection is equal to another collection `o`.
Returns the hash code for this collection.
Returns true if this collection contains no elements.
Removes the element `o` from this collection.
Removes all the elements in `c` from this collection.
Retains the elements that are both in `c` and in this collection.
Returns the number of elements in this collection.
Returns an array of `Object` for the elements in this collection.

Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

The List Interface

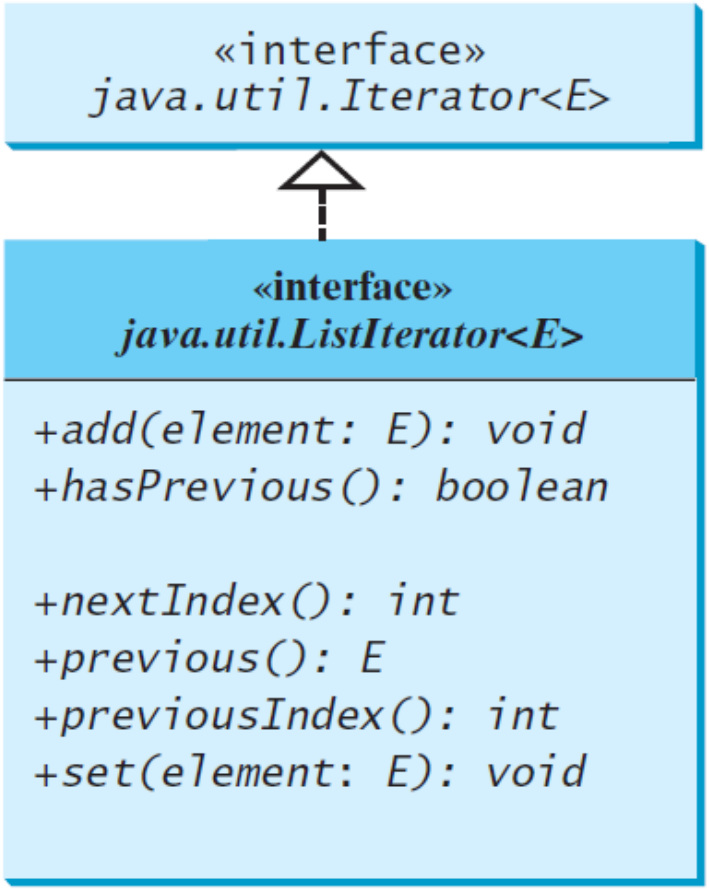
A list stores elements in a sequential order, and allows the user to specify where the element is stored. The user can access the elements by index.

The List Interface, cont.



`+add(index: int, element: Object): boolean`	Adds a new element at the specified index.
`+addAll(index: int, c: Collection<? extends E>) : boolean`	Adds all the elements in `c` to this list at the specified index.
`+get(index: int): E`	Returns the element in this list at the specified index.
`+indexOf(element: Object): int`	Returns the index of the first matching element.
`+lastIndexOf(element: Object): int`	Returns the index of the last matching element.
`+listIterator(): ListIterator<E>`	Returns the list iterator for the elements in this list.
`+listIterator(startIndex: int): ListIterator<E>`	Returns the iterator for the elements from `startIndex`.
`+remove(index: int): E`	Removes the element at the specified index.
`+set(index: int, element: Object): Object`	Sets the element at the specified index.
`+subList(fromIndex: int, toIndex: int): List<E>`	Returns a sublist from `fromIndex` to `toIndex-1`.

The List Iterator

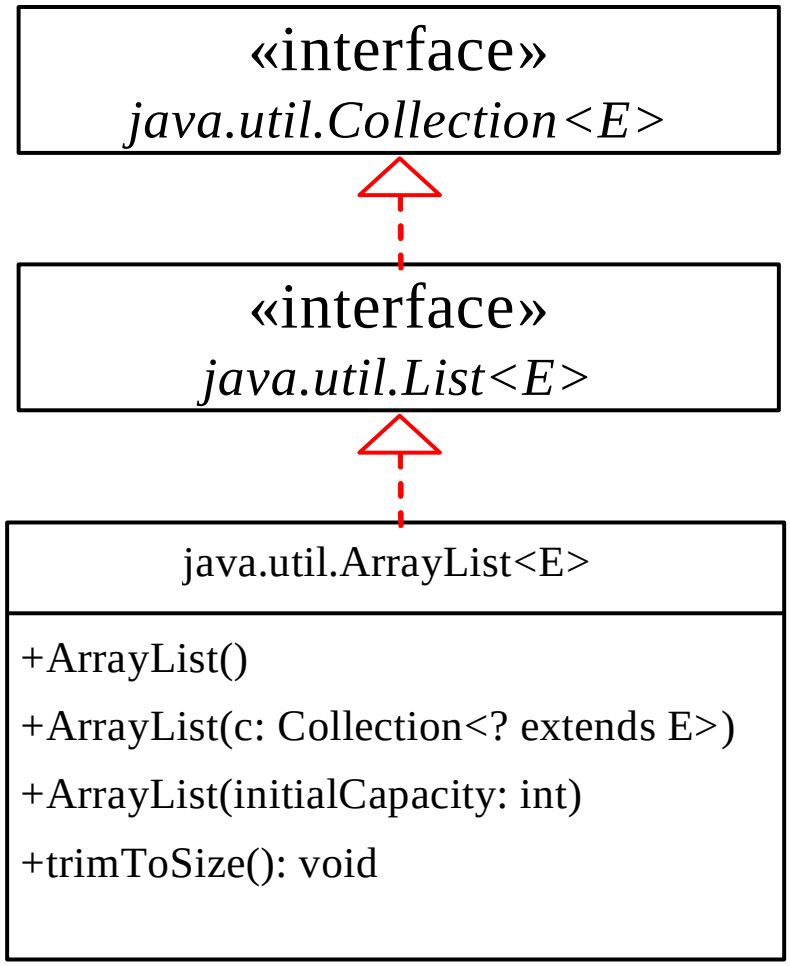


`+add(element: E): void`	Adds the specified object to the list.
`+hasPrevious(): boolean`	Returns true if this list iterator has more elements when traversing backward.
`+nextIndex(): int`	Returns the index of the next element.
`+previous(): E`	Returns the previous element in this list iterator.
`+previousIndex(): int`	Returns the index of the previous element.
`+set(element: E): void`	Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection. If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList



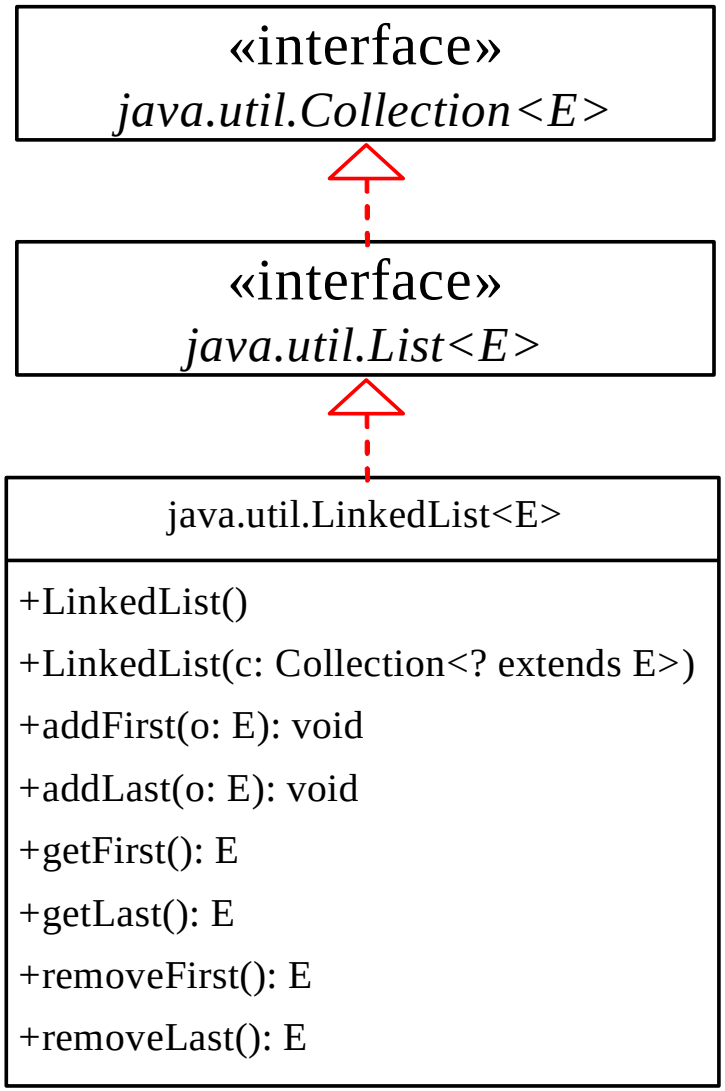
Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

Example: Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.

TestArrayAndLinkedList

Run

The Comparator Interface

Sometimes you want to compare the elements of different types. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, define a class that implements the java.util.Comparator interface. The Comparator interface has two methods, compare and equals.

The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

GeometricObjectComparator

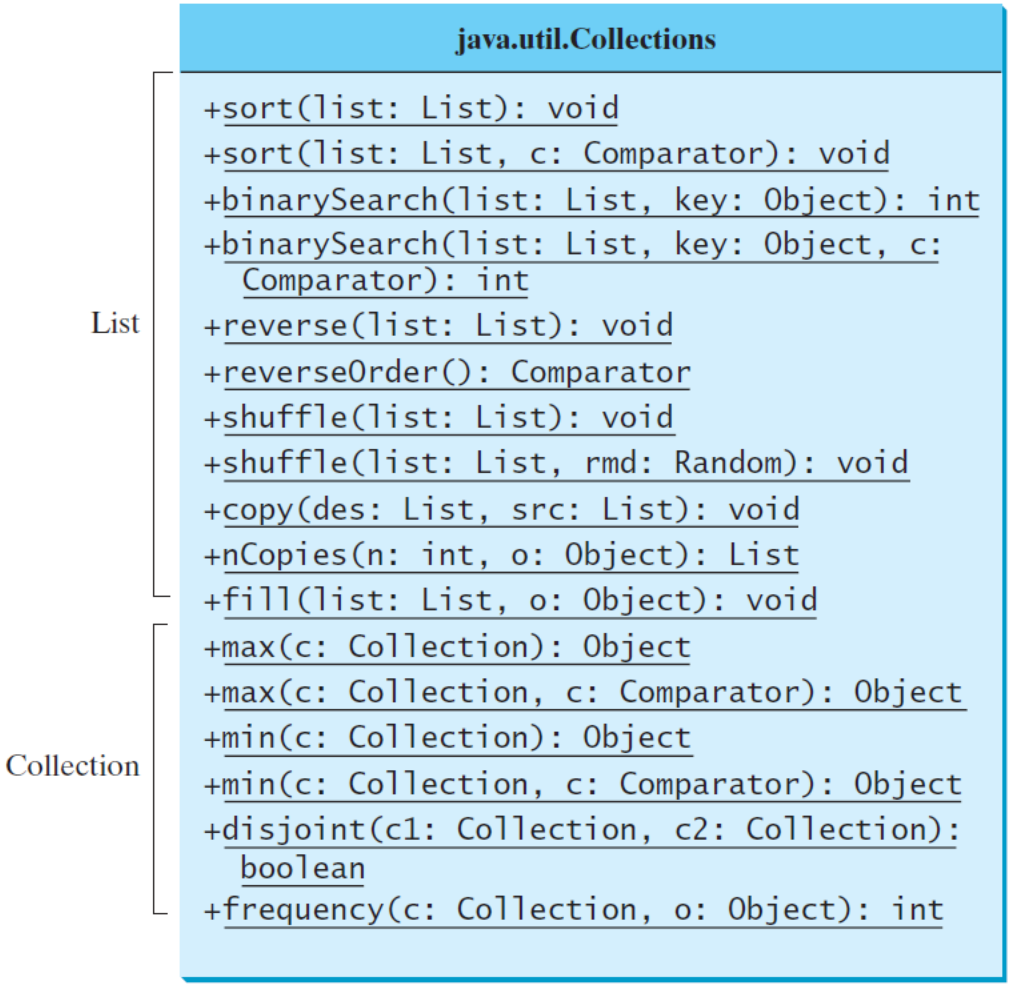
TestComparator

Run

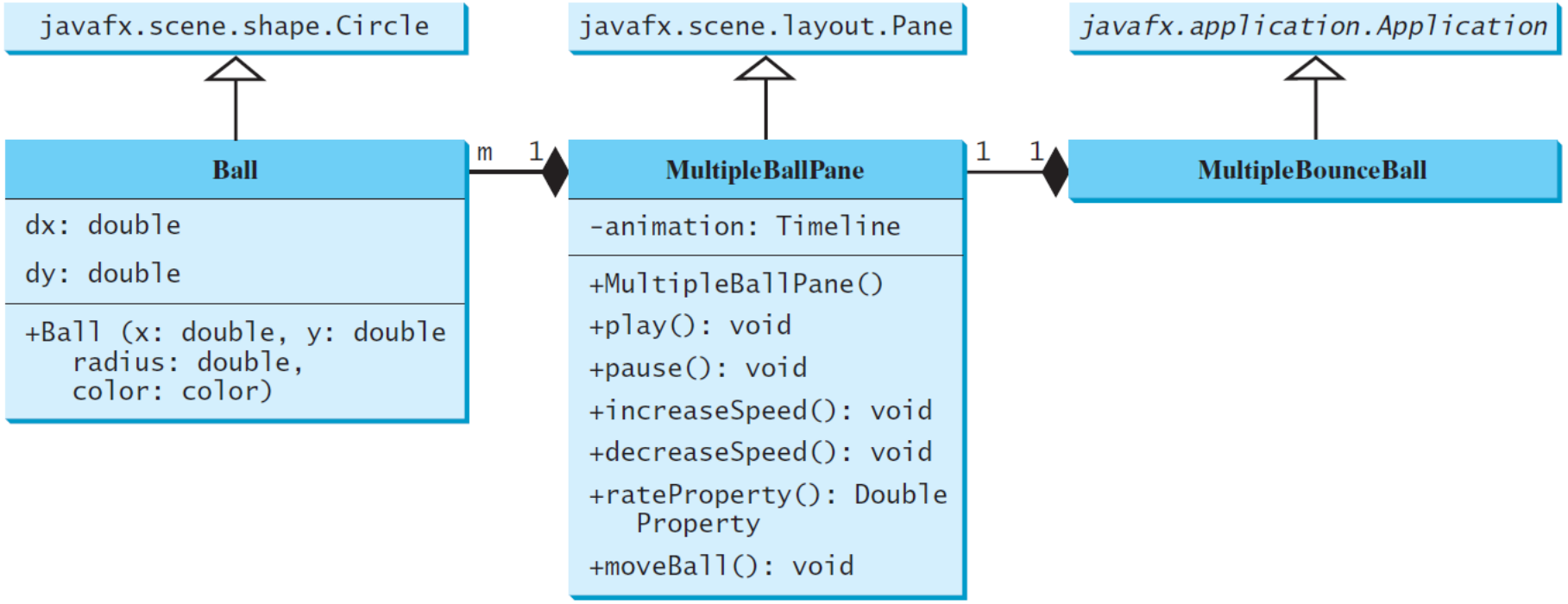
The Collections Class

The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

The Collections Class UML Diagram



Case Study: Multiple Bouncing Balls



MultipleBounceBall Run

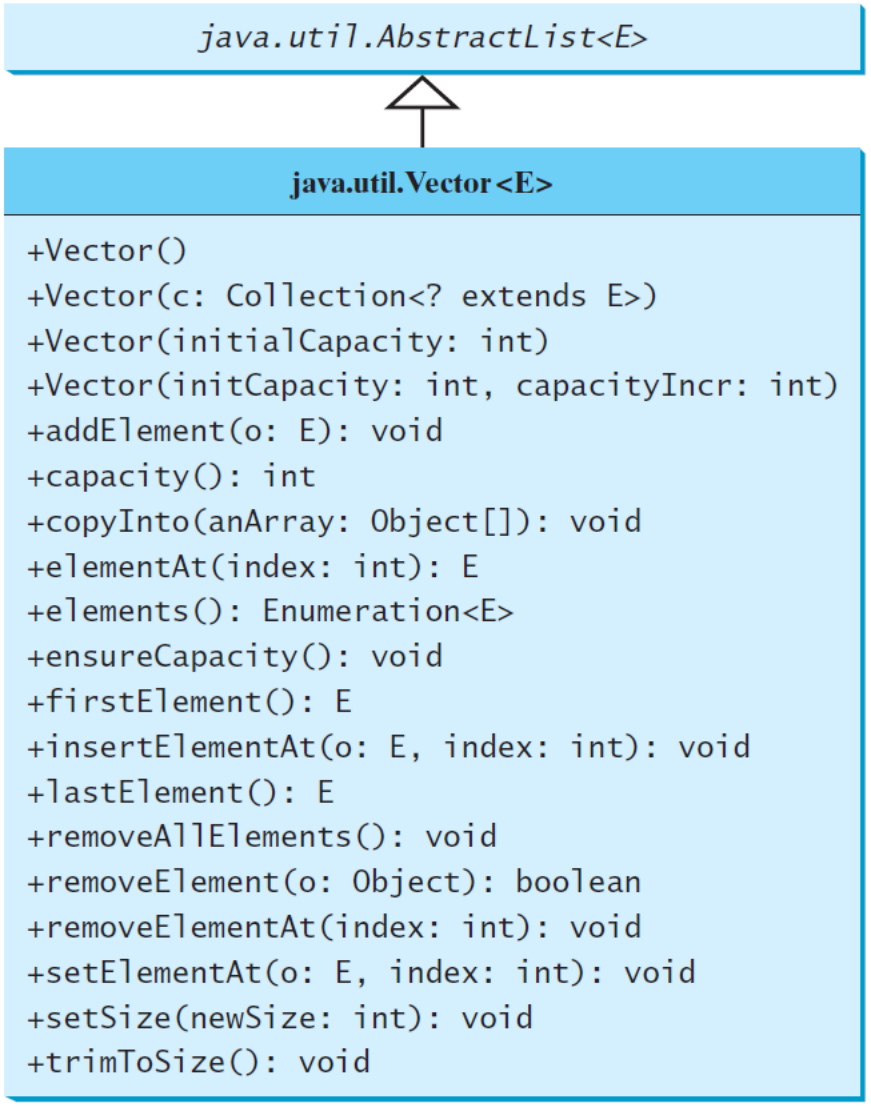
The Vector and Stack Classes

The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.

The Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, “The Collections Class.”

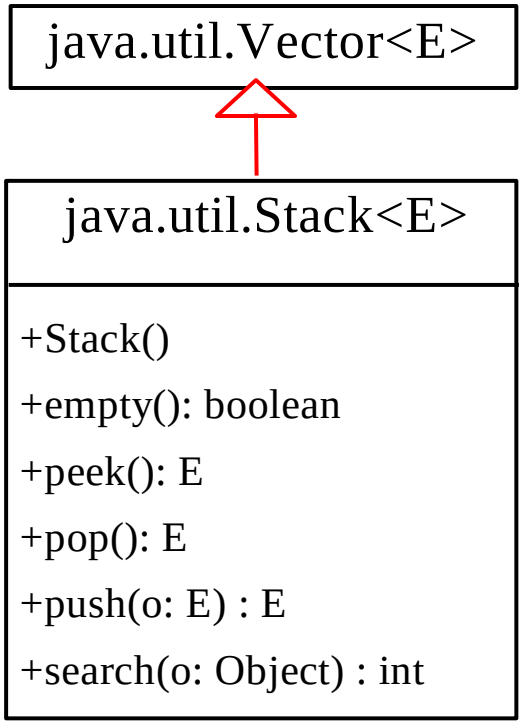
The Vector Class, cont.



Creates a default empty vector with initial capacity 10.	
Creates a vector from an existing collection.	
Creates a vector with the specified initial capacity.	
Creates a vector with the specified initial capacity and increment.	
Appends the element to the end of this vector.	
Returns the current capacity of this vector.	
Copies the elements in this vector to the array.	
Returns the object at the specified index.	
Returns an enumeration of this vector.	
Increases the capacity of this vector.	
Returns the first element in this vector.	
Inserts o into this vector at the specified index.	
Returns the last element in this vector.	
Removes all the elements in this vector.	
Removes the first matching element in this vector.	
Removes the element at the specified index.	
Sets a new element at the specified index.	
Sets a new size in this vector.	
Trims the capacity of this vector to its size.	

The Stack Class

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

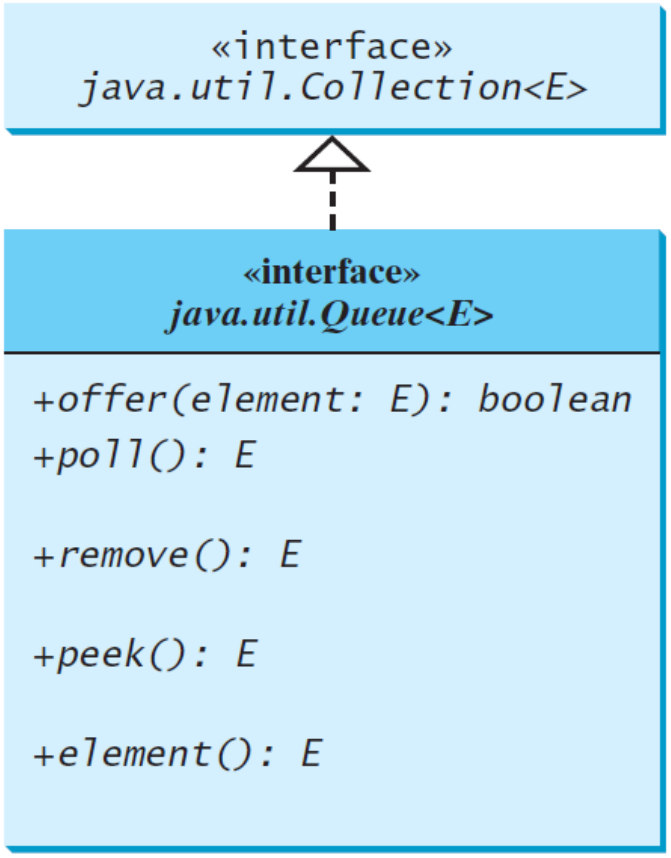
Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

Queues and Priority Queues

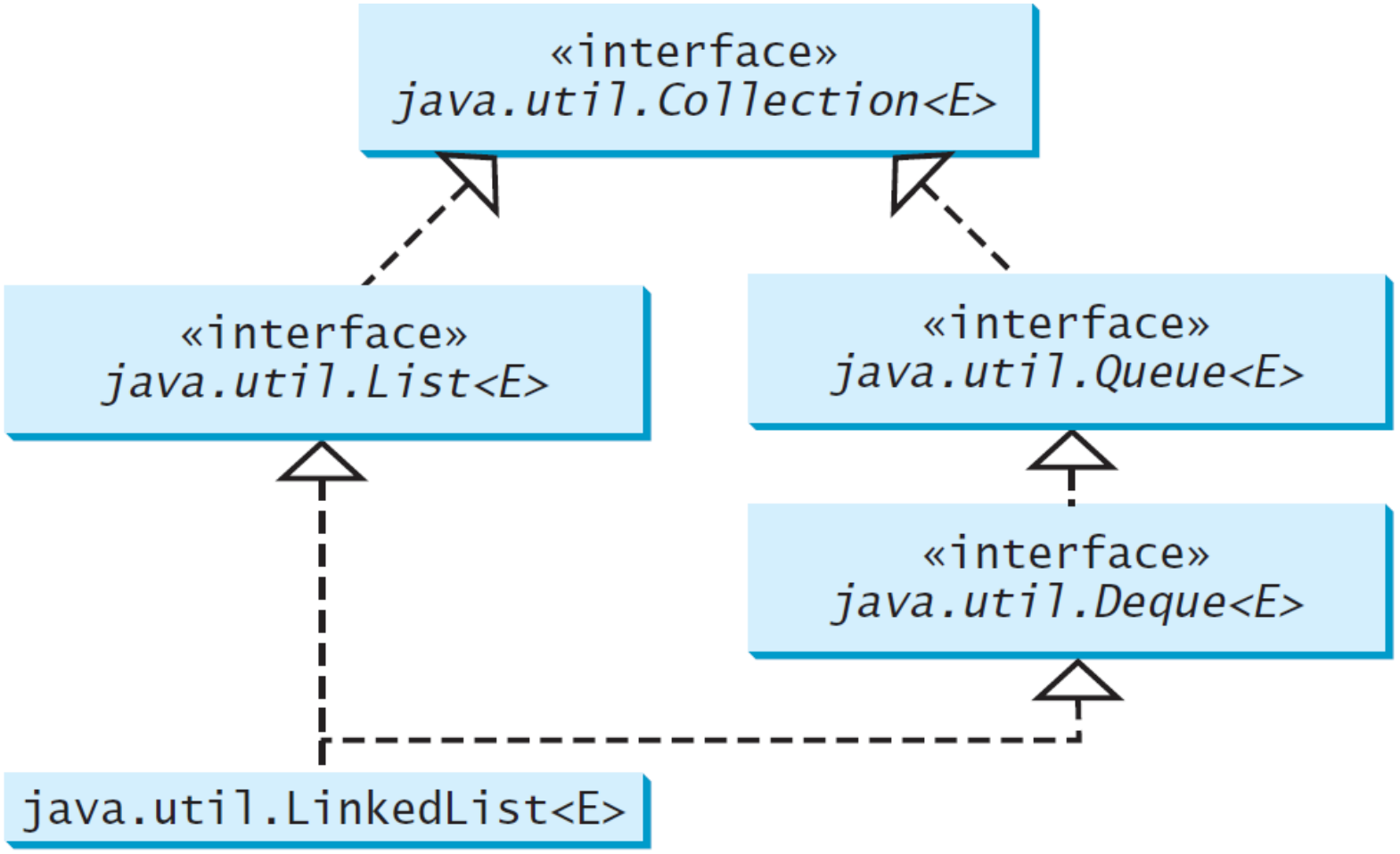
A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

The Queue Interface

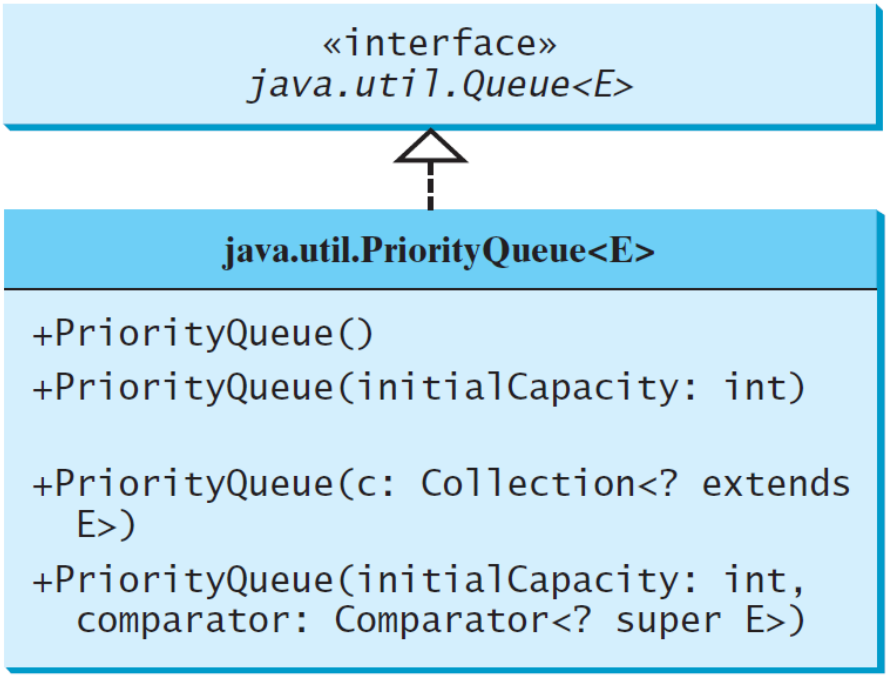


+offer(element: E): boolean	Inserts an element into the queue.
+poll(): E	Retrieves and removes the head of this queue, or `null` if this queue is empty.
+remove(): E	Retrieves and removes the head of this queue and throws an exception if this queue is empty.
+peek(): E	Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
+element(): E	Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

Using LinkedList for Queue



The PriorityQueue Class



Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

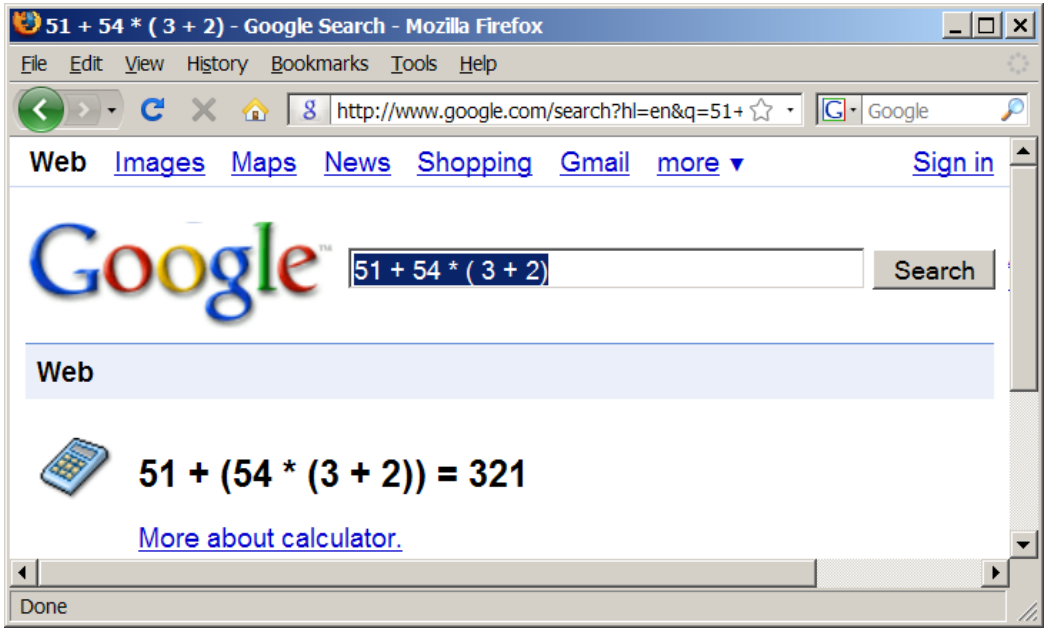
Creates a priority queue with the specified initial capacity and the comparator.

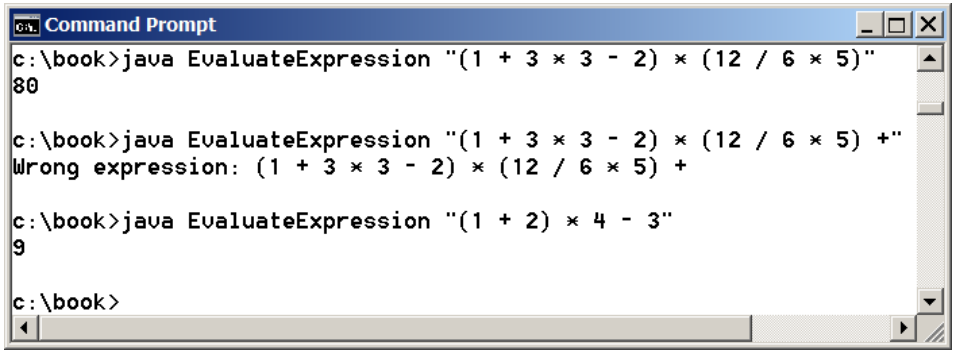
PriorityQueueDemo

Run

Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.





Algorithm

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a **+** or **-** operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a ***** or **/** operator, process the ***** or **/** operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a **(** symbol, push it to **operatorStack**.

1.5. If the extracted item is a **)** symbol, repeatedly process the operators from the top of **operatorStack** until seeing the **(** symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

Example

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3 ↑	(	Phase 1.4		(
(1 + 2)*4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2)*4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3 ↑	)	Phase 1.5	3	
(1 + 2)*4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2)*4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3 ↑	−	Phase 1.2	12	−
(1 + 2)*4 − 3 ↑	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3 ↑	none	Phase 2	9	